FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

ASUND: Solução de classificação estática em Node.js para aplicações JavaScript

António Cardoso Soares

DISSERTAÇÃO

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Miguel Pimenta Monteiro

11 de Julho de 2017

ASUND: Solução de classificação estática em Node.js para aplicações JavaScript

António Cardoso Soares

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Jorge A. Silva

Arguente: Fernando Mouta

Vogal: Miguel Pimenta Monteiro

---

11 de Julho de 2017

Resumo

JavaScript é uma das linguagens de programação mais populares no mundo. Devido à proliferação da sua aplicação a diferentes contextos têm surgido diversos problemas que têm sido abordados recentemente. A análise de código JavaScript é, há muito tempo, conhecida como um desafio em várias áreas devido à sua natureza dinâmica e, sendo uma linguagem interpretada, as aplicações podem ficar expostas a diferentes problemas de segurança. Para resolver alguns dos problemas foram desenvolvidas soluções usando a análise em tempo de execução ou aplicando diferentes técnicas de análise estática.

A criação de uma solução que consiga identificar as bibliotecas, assim como os diferentes contextos analisando apenas o código-fonte, poderá ter várias aplicações em marketing, vendas, na criação de um conjunto de dados que dê suporte a máquinas de aprendizagem automática e configuração ou seleção prévia de aplicações adaptadas ao contexto ou à presença de determinadas bibliotecas.

Assim, nesta dissertação, foi desenvolvida uma aplicação modular capaz de, a partir da análise estática de código-fonte, detetar ou inferir o uso de bibliotecas e o contexto para o qual foi desenvolvido. A solução foi dividida em três módulos, cada um responsável por ações distintas mas necessários para atingir os objetivos. As ações principais passam por recolher bibliotecas JavaScript presentes em repositórios *open-source* como o *GitHub* e fazer a recolha de indicadores para detetar o contexto e a *API*. Esses dados serão usados na deteção ou inferência de bibliotecas usadas, assim como o contexto de ficheiros ou aplicações JavaScript submetidas para avaliação.

A solução foi validada analisando a precisão e a sensibilidade do sistema através da submissão de projetos previamente classificados e quando possível, comparando as bibliotecas identificadas através de *metadata* e *imports* com os inferidos pelas chamadas a *API* externas.

Abstract

JavaScript is one of the most popular programming languages in the world. Due to its increasing use in different contexts, many problems have arisen that have been addressed recently. JavaScript code analysis has been regarded for some time as a challenge in several areas due to the languages' dynamic nature, and being an interpreted language, applications using JavaScript may be exposed to all kinds of security problems. To tackle some of these problems, solutions have been developed using techniques such as runtime analysis or static analysis.

Coming up with a solution that can not only identify the libraries, but also the contexts of an application by analysing the source code alone may have several uses in areas such as marketing, sales, building a set of data to support automatic learning machines, as well as the configuration, or previous selection of applications that fit the context or that play well with certain libraries.

Therefore, in this dissertation, a modular application was developed that is capable of detecting or inferring the usage of libraries and the context of a certain application, from the static analysis of its source code. The solution was divided into three modules, each of them responsible for different tasks but all necessary for the final goal. The main tasks were retrieving JavaScript open-source libraries hosted in public repositories such as GitHub, collection of indicators to detect the context, and the exported API. This data will be used on the detection and inference of libraries called, as well as the context of files or JavaScript applications submitted for evaluation.

The solution was validated by analysing its precision and sensitivity through the submission of previously classified projects, and when possible, by comparing the identified libraries through the metadata and imports with the ones that were inferred by external API calls.

Agradecimentos

Em primeiro lugar, à comunidade da FEUP, desde professores a colegas de estudo, que contribuíram durante vários anos para a minha evolução pessoal e profissional na área de engenharia informática. Em especial, ao professor Miguel Pimenta Monteiro pela orientação e dedicação que disponibilizou desde o primeiro dia.

À Jscrambler e toda a sua equipa por me receberem de braços abertos, e em especial ao Engenheiro Pedro Fortuna e Carlos Matias pelo apoio essencial à realização da dissertação.

Aos amigos e familiares, que me apoiaram nos bons e maus momentos nesta difícil etapa da minha vida.

Muito obrigado,
António Cardoso Soares

"If I have seen further it is by standing on the shoulders of Giants."

Isaac Newton

Conteúdo

Lista de Figuras

Lista de Listagens

Lista de Tabelas

Abreviaturas e Símbolos

API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
CLI	*Command-Line Interface*
IDE	*Integrated Development Environment*
IR	*Information Retrieval*
JS	JavaScript
JSON	*JavaScript Object Notation*
MSR	*Mining Software Repositories*
npm	*node package manager*
PR	*Pull Request*
RI	Recuperação de Informação
WWW	*World Wide Web*

Capítulo 1

Introdução

Este primeiro capítulo tem como propósito realizar a introdução ao tema deste documento, começando por fazer o seu enquadramento, explicando a motivação e identificando os principais objetivos necessários para a resolução do problema. Por fim, será apresentada a organização geral do documento.

1.1 Contexto

JavaScript (JS) é uma das linguagens de programação mais populares no mundo [sta17, Git17d]. A sua contínua evolução e proliferação a diferentes plataformas e áreas de elevada importância, como o comércio online [Pad13] ou instituições financeiras [Har13], levam à necessidade de aumentar a segurança e proteção nos produtos, serviços ou projetos que usem JS como linguagem. Sendo uma linguagem interpretada, o código-fonte pode ser facilmente acedido, permitindo assim expor possíveis vulnerabilidades nos sistemas. Para tentar diminuir ou remover este problema foram criadas diferentes soluções de proteção e otimização de código. Neste contexto, a empresa Jscrambler foi uma das empresas que desenvolveu vários processos para proteger diferentes aplicações. Um dos objetivos destas proteções, é blindar o código-fonte contra possíveis furtos e adulteração, sendo para isso necessário aplicar diferentes transformações ao código. A aplicação de transformações ou outro tipo de proteções podem ter algumas limitações. Por exemplo, um jogo ou uma aplicação desenvolvida para o uso mobile podem ter como limitações o seu tamanho e a performance, o que torna bastante importante evitar o impacto das proteções aplicadas nestas áreas.

No passado, foram desenvolvidas plataformas online e *plugins* que permitem detetar as tecnologias usadas por um sítio na *World Wide Web* (*WWW*). O uso dessas plataformas podem servir um conjunto de profissionais no ramo de engenharia de software ou *marketing*, mas para já são limitadas à analise dinâmica de sítios publicados na *WWW*.

Uma das dificuldades atuais é a identificação automática do contexto e conteúdo de aplicações ou partes de aplicações através da análise estática, principalmente se estes não estiverem expostos através de *metadata* ou *imports*.

1.2 Motivação e Objetivos

Com a recente evolução do ecossistema e intenção de publicação anual de novos padrões de JS, estes nem sempre são logo aplicados pelos navegadores e plataformas, principalmente em dispositivos móveis ou sistemas operativos antigos e desatualizados. Por exemplo, os programadores tendem a usar especificações antigas para que não surjam erros indesejados. Atualmente, também é possível desenvolver todas as partes de uma aplicação apenas usando JS, produzindo assim aplicações JS com partes que se aplicam a contextos completamente distintos.

A proteção do código-fonte de uma aplicação pode ser vital. As empresas ou os programadores de aplicações inovadoras ou que manipulam dados sensíveis necessitam de as blindar contra possíveis roubos, adulteração ou exposição de dados. As diferentes proteções podem beneficiar de toda a informação possível sobre uma aplicação. Por exemplo, um jogo não pode ter a sua performance afetada, por isso deve-se evitar a aplicação de transformações no código que possam deteriorar a sua jogabilidade e mantendo o objetivo de reduzir ao máximo o impacto na aplicação das proteções necessárias.

A informação nunca é suficiente, pois é uma ferramenta útil e fundamental no planeamento de uma equipa de engenheiros de software, vendas e *marketing*. Informações como o contexto e a identificação do uso de bibliotecas numa aplicação podem fornecer dados necessários para mudar uma abordagem, o plano de desenvolvimento ou a aplicação de uma estratégia.

Para tal, é necessário desenvolver a capacidade de automaticamente identificar que *frameworks*, módulos e bibliotecas uma aplicação usa e para que contexto (*server*, *front-end*, *mobile*, entre outros) foram desenvolvidas. Esta tarefa não é trivial, se não estiverem declarados em *imports* ou *metadata*, sendo assim necessário usar a partir da análise do código estes e outros indicadores que ajudem na identificação.

Pretende-se com esta dissertação desenvolver um processo de classificação em *Node.js*, capaz de identificar a presença de *frameworks* e *bibliotecas* assim como o seu contexto em aplicações JS, analisando o seu código fonte.

Para isso, será necessário recolher, de repositórios de larga escala, uma série de indicadores de bibliotecas, como por exemplo as suas funções públicas que servirão para a construção de uma base de dados de conhecimentos, que, juntamente com as indicações presentes no código fonte dos ficheiros, servirá para identificar o uso de bibliotecas e inferir o seu contexto.

1.3 Benefícios Esperados

Espera-se que a identificação de bibliotecas e *frameworks* presentes em aplicações, assim como os diferentes contextos detetados diretamente ou por inferência, possam a vir ter várias aplicações. Esta é uma tarefa difícil quando apenas se tem acesso ao código-fonte e não é garantido o acesso a *metadata* ou *imports*.

Nesse sentido espera-se que a solução apresentada nesta dissertação seja capaz de identificar estaticamente a presença de *frameworks*, bibliotecas e o contexto de uma aplicação JS ou suas partes, analisando apenas o código-fonte. Esta solução deverá permitir melhorar e até automatizar o processo de seleção do modo de proteção da empresa Jscrambler, para que através da identificação do contexto se apliquem as parametrizações adequadas. Também se pretende que os resultados ajudem possíveis estudos académicos ou empresariais que necessitem extrapolar a informação a partir deste tipo de dados. Por exemplo, o conjunto de dados recolhidos, poderão servir de apoio ao desenvolvimento de uma máquina de aprendizagem para detetar o contexto.

1.4 Estrutura da Dissertação

Para além deste primeiro capitulo onde é realizada uma introdução ao tema e contexto desta dissertação, este documento contém mais 4 capítulos.

No capítulo 2, é realizada uma revisão bibliográfica acerca das definições e conceitos que servem de base a esta dissertação. São ainda apresentados trabalhos relacionados e tecnologias importantes à realização da solução.

No capítulo 3, é apresentada uma descrição detalhada dos requisitos, arquitetura e implementação.

No capítulo 4, são apresentados e discutidos os resultados.

Por último, no capítulo 5, são retiradas as principais conclusões e descritas algumas orientações sobre o trabalho futuro.

Capítulo 2

Background e Estado da Arte

Neste capítulo são apresentadas as principais definições e conceitos, assim como trabalhos relacionados, tecnologias e componentes que servem de base à realização desta dissertação.

2.1 JavaScript

JavaScript (JS) é uma linguagem de programação interpretada, que foi originalmente desenvolvida por Brendan Eich [Int17b]. Esta adquiriu mais tarde uma ampla aceitação como a linguagem de *scripting* para o *frontend* das páginas web, aumentando a sua popularidade. Em novembro de 1996 a Netscape anunciou que tinha submetido o JS para a ECMA *international* como candidato a padrão industrial. O trabalho subsequente resultou no padrão que atualmente é conhecido como *ECMAScript*. Tem ocorrido uma evolução da linguagem, com contínuas publicações de novas edições por parte do grupo TC39-TG1, sendo a última designada *ECMAScript* 2017[Int17a]. Neste momento, já existe outra proposta com o nome de *ECMAScript* 2018[Int17b] e tudo aponta para que seja finalizada e publicada durante o próximo ano.

2.1.1 Aplicações

A evolução da linguagem JS, a proliferação de *frameworks* e bibliotecas, as práticas de programação melhoradas, os rápidos interpretadores e as novas funcionalidades, que são introduzidas a cada nova edição, contribuíram para que, deixasse de estar apenas limitada a navegadores e se expandisse a outros contextos como servidores[Dev17], dentro de outras aplicações como por exemplo em bases de dados, editores de motores de jogo[Tec17] e até para criar aplicações em diferentes sistemas operativos usando para isso motores de *scripting* como V8[Dev17] ou o *JavaScriptCore*[App17].

O resultado é a possibilidade de, usando apenas uma linguagem, desenvolver todos os pontos cruciais e necessários para a criação de aplicações ricas e complexas. Isto permite aumentar a liberdade dos programadores, que antes estavam limitados a uma área de desenvolvimento, mas sem grande esforço conseguem entrar noutras áreas de desenvolvimento e tornarem-se programadores *fullstack*, usando, por exemplo, múltiplas soluções criadas como a *MEAN stack* [Len14].

2.1.2 Segurança e a solução Jscrambler

JavaScript é uma linguagem cada vez mais poderosa, porém no que toca a segurança, tem o problema de ser uma linguagem interpretada. Linguagens compiladas geram código binário antes de serem executadas, o que fornece uma proteção extra contra leitura e alteração indevida do código-fonte. Porém como o JS é interpretado e executado em *runtime*, todo código pode ficar exposto, tornado acessível a humanos ou máquinas com intenções maliciosas. É possível visualizar nos navegadores todo o código-fonte aí executado por uma aplicação. Também é possível aceder e manipular informações sensíveis de um telemóvel ou *desktop* como a câmara, o GPS, o *bluetooth*, o *wifi*, entre outras funcionalidades, que *frameworks* como o *PhoneGap*[Pho17] ou *Electron*[Git17a] fornecem ao programador e que também ficam acessíveis no código-fonte.

Para assegurar que uma aplicação não é adulterada, é necessário aplicar técnicas de ofuscação resistentes, com camadas adicionais de segurança.

A Jscrambler é uma empresa com milhares de clientes, incluindo empresas *Fortune* 500 e que disponibiliza um serviço que inclui uma solução completa para proteção de código JS multiplataforma, ou seja, proteção de integridade para aplicações *mobile*, *desktop*, *web* e *server-side*, evitando diminuir a performance ou prejudicar a funcionalidade de uma aplicação. Os seus serviços possuem diferentes funcionalidades de proteção como ofuscação, minimização, ocultação de dados sensíveis, prevenção de adulteração e limitação do código a uma plataforma ou a um determinado período de tempo. Estes podem ser selecionados pelo utilizador do serviço e podem depender do conteúdo da aplicação.

É intenção usar o resultado desta dissertação para diferentes aplicações, como permitir automatizar processos na submissão de aplicações na plataforma de proteção, desenvolver estudos e estratégias em inteligência empresarial e permitir melhorar a proteção das aplicações aplicando diferentes modos às diferentes partes da aplicação. O objetivo será melhorar a experiência do utilizador e o desempenho geral da solução.

2.2 Prospeção de dados em repositórios de código

A prospeção de dados é uma área que tem vindo a ser estudada há vários anos e recentemente, aplicada a repositórios de software, levou à criação de uma área de estudo chamada *mining software repositories*(MSR). Esta analisa e cruza dados para encontrar informação útil sobre sistemas de software e projetos[Has08]. Repositórios de histórico, execução e código são três exemplos de repositórios geralmente usados nos estudos, sendo os repositórios de código, que contêm o código-fonte de *frameworks* e bibliotecas, o alvo de discussão desta dissertação.

Repositórios de código

O desenvolvimento de aplicações modernas envolve geralmente um grande número de programadores ou equipas de programadores, que muitas vezes trabalham em localizações e horários

diferentes. A comunicação e a coordenação de tais projetos necessita de diferentes meios de suporte que vão desde o controlo de versões do código a registo de erros e documentação[PSB11]. Estas plataformas, permitiram também que diferentes comunidades se juntassem e em conjunto trabalhassem na resolução de vários problemas que eram comuns a todos. Esse trabalho é disponibilizado utilizando licenças *open-source* para que qualquer programador possa usufruir e sentir-se à vontade para contribuir. Várias plataformas que suportam este tipo de atividade surgiram, como por exemplo, o *CodePlex* suportado pela Microsoft[Mic17], *GoogleCode* que deixou de ser suportado em 2016 pela Google[Goo17], *Sourceforge*[Med17] e *Github*[Git17c].

Segundo [WSR16, Spr17], *npm* e *bower* são os *package managers* mais populares na área de *open-source*. Depois de uma breve análise dos *packages* mais populares, verifica-se uma tendência na comunidade JS em adotar o *GitHub* como o repositório de referência. Contendo mais de 35 milhões de projetos este tem sido alvo de vários estudos de prospeção de dados [CLC16] e uso intensivo por parte da comunidade JS. Devido a estas razões este será usado como ponto de partida para a extração de código-fonte de bibliotecas e *frameworks* JS podendo usar como critério a popularidade, *commits* ou pontos *star*.

Estrutura de um repositório

Apesar de não existir uma norma oficial, os repositórios tendem a seguir uma convenção, seguindo uma típica organização com a seguinte estrutura [Git17b, ZZM14]:

build	Pasta com ficheiros compilados (alternativamente *dist*)
docs	Documentação (alternativamente *doc*)
src	Código-fonte (alternativamente *lib* ou *app*)
test	Testes automáticos (alternativamente *spec* ou *tests*)
tools	Ferramentas e utilidades
package.json	Ficheiro de configuração para pacotes *npm*
LICENSE	Licença
README.md	Descrição do projeto

Apesar dos repositórios com os projetos mais populares, ou seja, com mais *stars* e *forks*, já sigam na sua grande maioria esta convenção ou tenham usado as ferramentas *CLI* existentes para proceder à criação do esqueleto organizacional dos projetos, infelizmente ainda há vários que por desleixo dos seus criadores, ou pela sua antiguidade, seguem a sua própria estrutura. Isto torna a análise de tal repositório difícil e aumenta a complexidade da heurística usada para filtrar os ficheiros que devem ser analisados para que não se polua a base de conhecimentos com dados inválidos ou desnecessários. Outra dificuldade inerente da falta de organização comum é a distinção entre o que é um repositório que contém o código-fonte de uma biblioteca ou *framework* de um simples projeto, tutorial ou manual de boas práticas, projetos que são cada vez mais populares como por exemplo os projetos: *FreeCodeCamp*[Fre17], *You-Dont-Know-JS*[Sim17] e *Airbnb JavaScript Style Guide*[Air17].

Topics

Uma nova funcionalidade desenvolvida por um dos repositórios, o *GitHub*, são os *Topics*. Esta consiste em atribuir um contexto ou rótulo ao repositório para ajudar a comunidade a encontrar mais facilmente repositórios com assuntos e tópicos semelhantes ou, no caso deste projeto, poderá ser usado para identificar o contexto e conteúdo. A atribuição destes tópicos é inserida manualmente ou então sugerida através de uma máquina de aprendizagem automática e processador de linguagem natural que analisa o conteúdo do projeto. Cabe aos administradores do projeto aceitar ou rejeitar, sendo o resultado usado como treino. As desvantagens deste sistema é que os tópicos não são previamente definidos, ou seja, os administradores podem inserir manualmente qualquer palavra ou iniciais. O facto de este campo não ser obrigatório e até à data não ter suporte da *API* em produção, torna o seu uso pouco recomendado.

2.3 Análise estática do código

Análise estática em software consiste na análise de um programa de computador sem a necessidade de executar o código, sendo usado o código fonte para proceder a essa análise. Tem variadas aplicações, como encontrar e realçar erros, testar a qualidade de software crítico ou aplicar transformações que permitam compilação, ofuscação, proteção, e deteção de vulnerabilidades presentes no código.

O processo consiste em transformar um conjunto de texto, isto é, código-fonte, numa estrutura de dados, geralmente designada por árvore sintática abstrata (*AST*), usando um conjunto de regras predeterminado. Existem dois tipo de estratégias geralmente usadas: *top-down* e *bottom-up*.

A estratégia *top-down* consiste numa derivação mais à esquerda a partir do símbolo inicial de uma gramática. A árvore é construída da raiz até às folhas, onde para cada vértice são selecionados símbolos não terminais mais à esquerda onde os vértices filhos são os símbolos à sua direita. Este processo terminará quando todas as folhas forem símbolos terminais.

A estratégia *bottom-up* realiza uma redução mais à esquerda a partir do símbolo inicial da gramática. A árvore gramatical é construída iniciando pelas folhas e indo em direção à raiz. Os símbolos são associados até se reconhecer o lado direito de uma produção e a aceitação dá-se até estar esgotada a sequência e o símbolo inicial estiver na raiz da árvore.

JS é um desafio nesta área devido a ser uma linguagem muito dinâmica e híbrida. Apesar dos últimos desenvolvimentos, o estado da arte da análise estática de JS ainda continua muito atrasado em relação a outras linguagens como C ou Java. Porém tem existido um aumento de estudos direcionados às caraterísticas peculiares da linguagem e ao desenvolvimento de ferramentas. Por exemplo, o facto de existirem bibliotecas escritas em C++ aumenta a dificuldade de análise das mesmas, um dos estudos utiliza o uso de bibliotecas em código-fonte de diversas aplicações, para que se aumente os conhecimentos sobre a sua estrutura e funcionalidades [MLF15].

Para ajudar a identificar a presença de bibliotecas em código fonte e o contexto, uma série de indicadores deverão ser recolhidos através da analise estática do código como, por exemplo, a definição de funções públicas expostas para uso externo ou a presença de invocações de determinados serviços.

2.3.1 Árvore sintática abstrata

As Árvores sintáticas abstratas ou em inglês *Abstract Syntax Trees* (*AST*) são estruturas de dados em árvore que representam estruturas de um código-fonte escrito numa linguagem. A sintaxe é abstrata no sentido que ela não representa cada detalhe que aparece na sintaxe real. Por exemplo, agrupar parênteses está implícito na estrutura da árvore e uma construção sintática como um condicional pode ser representada por um simples nó com as suas ramificações.

A figura 2.1 mostra como uma linha de código contendo apenas texto é convertido numa estrutura *AST*, que neste caso é representado no formato *JavaScript Object Notation* (JSON). Formato geralmente escolhido, pois facilita a navegação e exploração da árvore em aplicações JS.

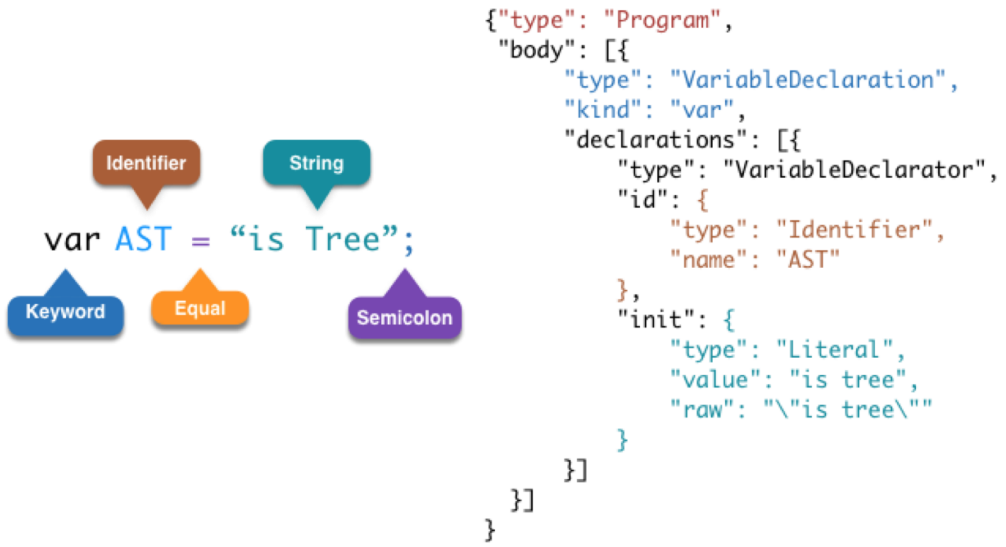

Figura 2.1: Conversão de código-fonte numa *AST*

Especificação *ESTree*

Existem diferentes especificações de árvores sintáticas, sendo uma delas, o *ESTree*[EST17], criada e mantida pela comunidade JS, tendo como principais responsáveis membros importantes de comunidades ou projetos *open-source* como Mozilla, *Acorn*, *Esprima* e Babel[EST17]. A especificação *ESTree* é uma expansão e daí a sua compatibilidade com a versão do *Mozilla Parser API*[EST17].

2.4 *Information Retrieval*

Information Retrieval (*IR*), ou em português recuperação de informação (RI), é geralmente considerada uma área das ciências de computadores que lida com a representação, armazenamento e acesso a informação, ou seja, RI é um processo pelo qual uma coleção de dados é representada, armazenada e pesquisada com o intuito de fornecer conhecimento em resposta a uma consulta.

Os sistemas criados usando este processo são geralmente aplicados a bibliotecas digitais, motores de busca ou pesquisa multimédia [SP13]. Para atingir o seu objetivo, os sistemas RI geralmente seguem este processo apresentado na Figura 2.2 [SP13].

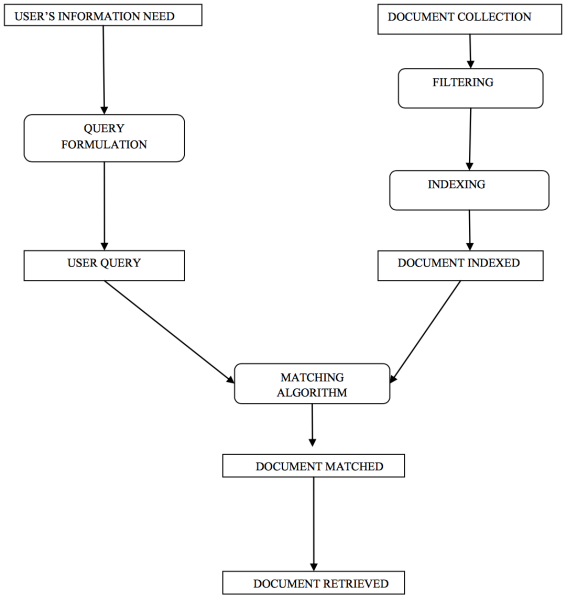

Figura 2.2: Modelo típico de RI

Uma forma de abordar o problema desta dissertação será aplicando um modelo RI semelhante ao de um motor de busca onde os documentos seriam as bibliotecas previamente analisadas e onde se indexaria a *API* exposta e os termos de pesquisa a invocações dessas bibliotecas. O resultado deverá ser um conjunto de bibliotecas identificadas que depois por inferência se usará para detetar o contexto.

2.4.1 Modelos RI

Os modelos RI descrevem os detalhes da representação, pesquisa e recuperação dos documentos. Os principais modelos podem ser classificados como booleanos, vetoriais e probabilísticos.

2.4.1.1 Booleanos

O modelo booleano permite o uso de operadores lógicos booleanos como *AND*, *OR* e *NOT* para a formalização da pesquisa. Uma das suas desvantagens é não ser capaz de classificar o

resultado retornado, ou seja, classifica-os como relevantes ou irrelevantes[Alh10, SP13].

2.4.1.2 Vetoriais

O modelo vetorial tem como principal caraterística ordenar por similaridade os documentos com os termos pesquisados. Estes são representados por vetores e o ângulo entre os dois é calculado usando a função semelhança do cosseno.

2.4.1.3 Probabilísticos

O modelo probabilístico ordena os documentos pela sua probabilidade de relevância. Os documentos e as pesquisas são representados por vetores binários sendo que cada elemento indica se os termos ou atributos da pesquisa estão presentes no documento e vice-versa. Este modelo em vez de usar o valor da probabilidade usa *odds* onde $O(R) = P(R)/1 - P(R)$, sendo R a relevância do documento.

2.4.2 Técnicas de Indexação

Existem várias técnicas de indexação onde se destacam as listas invertidas e as *signature files*. As listas invertidas associam documentos aos índices tornando o processo de pesquisa de termos mais rápidos, enquanto que as *signature files* armazenam dados ou assinaturas do documento facilitando assim a sua a pesquisa.

2.4.3 Técnicas de pesquisa

Existem vários algoritmos onde se incluem as seguintes técnicas:

- Pesquisa linear ou sequencial, procura um elemento particular de uma lista ou *array*. São percorridos todos os elementos um de cada vez. É um simples algoritmo que tem a desvantagem de ser lento pois é sequencial[SP13].

- Pesquisa binária que consiste na procura de um elemento num *array* ordenado. A cada passo este compara o valor chave com o meio do *array*. Caso este seja o valor procurado retorna-o. No caso contrário, se o valor for mais pequeno volta-se a repetir o processo na metade inferior do *array* ou na superior se o valor for maior. Este algoritmo permite pesquisas mais rápidas mas o custo de ordenação é bem mais pesado, sendo também difícil ordenar os termos[SP13].

- A pesquisa de força bruta consiste na enumeração de todos os possíveis candidatos para uma solução e verifica se cada candidato satisfaz a condição de resolução do problema. Este é um algoritmo simples que encontra sempre a solução, caso esta exista[SP13].

2.4.4 Validação

Em *IR* a validação pode ser feita através da precisão e sensibilidade, também conhecida como *recall*. Ambas são importantes para a medida de relevância.

Nesta área, a precisão resume-se a quantos elementos retornados pela pesquisa são relevantes ou seja verdadeiros positivos, enquanto que sensibilidade mede quantos elementos relevantes foram selecionados, tal como se pode observar na figura 2.3.

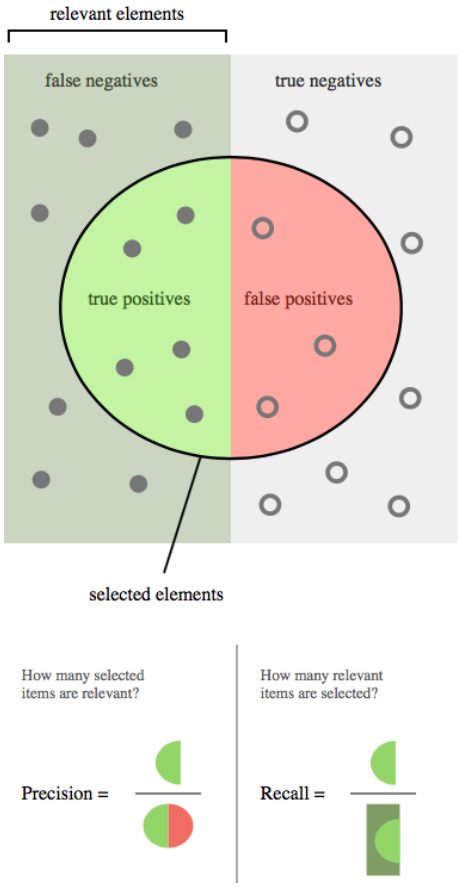

Figura 2.3: Precisão e sensibilidade [Wal17]

Por exemplo, se considerarmos um mundo onde existam 8 documentos sobre gatos e 2 sobre cães, se pesquisarmos por documentos que contenha gatos e o resultado for 1 documento sobre cães e 6 sobre gatos. A precisão será $6/7$ e a sensibilidade $6/8$.

Resumindo, uma precisão alta pode ser vista como uma medida de qualidade onde são devolvidos mais resultados relevantes que irrelevantes, enquanto que sensibilidade alta significa que o sistema devolveu a maioria dos resultados relevantes. [POW11]

É possível combinar a precisão com sensibilidade numa nova medida chamada Valor-F. Esta consiste na média harmónica que combina os valores da precisão com sensibilidade da seguinte forma:

$$F = 2 \times \frac{precision \times recall}{precision + recall}$$

2.5 Trabalhos relacionados

Existem atualmente diferentes ferramentas para a identificação de tecnologias usadas em *websites* mas que produzem a sua avaliação através da leitura de *metadata* ou procura de caraterísticas específicas em *runtime*. Existem também outras ferramentas que usam técnicas de *machine learning* e extração de conhecimentos para diferentes meios como a deteção da *API* de uma biblioteca usando exemplos presentes no código-fonte, a recomendação de bibliotecas semelhantes e a sua utilização por outros utilizadores. Nesta secção serão descritas algumas delas.

2.5.1 *Wappalyzer*

Wappalyzer é uma pequena aplicação de deteção de tecnologias em *websites*. Está limitada aos navegadores mais populares em forma de *plugin*, usando técnicas de pesquisa por caraterísticas introduzidas manualmente e que identifiquem em *runtime* as bibliotecas usadas[Ali17].

2.5.2 BuildWith

Plataforma online que identifica tecnologias usadas por diferentes *websites*. Esta dispõe de uma *API* de acesso aos dados previamente indexados pelos seus indexadores automáticos[bui17].

2.5.3 *MAPO*

MAPO é uma ferramenta de recomendação de trechos de código-fonte extraídos de repositórios usando um processo automático de extração de padrões de uso de *APIs* de bibliotecas. Com uma simples pesquisa, como por exemplo o nome de uma função, classe, ou pacote de uma biblioteca, este fornece trechos de código-fonte de repositórios relevantes para análise. Esta ferramenta foi desenvolvida sobre uma *framework* de extração do uso de *APIs* que foi desenvolvida pelos mesmo autores[ZXZ+09].

2.5.4 *Sourcegraph*

Sourcegraph é um IDE pago, desenvolvido pela empresa *Sourcegraph*, que usa um motor de pesquisa de código que permite aos utilizadores procurar por qualquer função, tipo ou módulo em repositórios para verificar como os outros programadores desenvolvem. O estudo sobre o uso de *APIs* permite aos programadores melhorar a sua produtividade, reutilizar código e melhorar a qualidade do seu código[Med17].

2.6 Tecnologias e componentes apropriados à solução

2.6.1 Aplicações de análise estática

Para construir uma base de conhecimento e classificar as aplicações, será preciso fazer uma análise léxica (*tokenization*) e subsequente análise sintática (*parse*), extraindo os indicadores necessários, como por exemplo os nomes de objetos, funções declaradas, os seus argumentos e invocações.

Uma crença popular na área de engenharia de software, especialmente na comunidade *open-source*, é que se deve evitar reinventar a roda e nesse sentido foi realizada uma pesquisa por aplicações que executem estas tarefas. Existem variadas ferramentas e aplicações de análise mas apenas serão destacadas as que mais se enquadram no tema deste trabalho, isto é, que sejam facilmente adaptadas ou funcionem em *Node.js*, e que forneçam suporte à análise de código-fonte usando os últimos padrões do *ECMAScript*.

2.6.1.1 Esprima

Esprima é um *parser* de alto desempenho, compatível e escrito com o padrão ECMAScript [Hid17]. Destaca-se as seguintes caraterísticas:

- Suporta de *Node.js*.
- Total suporte para o padrão *ECMAScript* 2016 (ECMA-262 7th Edition).
- Resultados em forma de *token* ou estrutura *AST* usando o padrão *ESTree*.
- Bem testado com uma média de 1500 testes sobre a totalidade da sintaxe.
- Licença *BSD license*.

2.6.1.2 *Espree*

Espree foi um *fork* do *Esprima* antes do suporte em *ECMAScript* 2016 ser desenvolvido mas que agora usa o *Acorn* como base [ESL17]. As suas caraterísticas mais relevantes são:

- Suporta *Node.js*.
- Suporte para o padrão *ECMAScript* 2016 (ECMA-262 *7th Edition*).
- Contém testes unitários.
- Resultados em forma de *token* ou estrutura *AST* usando o padrão *ESTree*.
- Licença *BSD 2-clause*.

2.6.1.3 *Acorn*

Acorn é um *parser* de *ECMAScript open-source* criado por Marijn Haverbeke e bastante usado como base em diferentes projetos. Este dispõe de uma arquitetura modular com suporte de *addons*, que permite estender ou melhorar as suas funcionalidades principais [Hav17]. As caraterísticas que se podem destacar são:

- Suporta *Node.js*.
- Tem suporte para o padrão *ECMAScript* 2016 (ECMA-262 *7th Edition*).
- Bem testado, contendo diferentes testes baseados no *Esprima*.
- Retorna os resultados em forma de *token* ou estrutura *AST* usando o padrão *ESTree*.
- Suporte de *plugins*
- Licença *MIT*

2.6.1.4 *Babylon*

Babylon é um *parser* de *ECMAScript* baseado no *Acorn* e usado pelo JavaScript *compiler* Babel. As suas principais caraterísticas são:

- Suporta *Node.js*.
- Tem suporte para o padrão *ECMAScript* 2016 (ECMA-262 *7th Edition*).
- Aceita *PRs* de propostas experimentais em pelo menos *stage* 0.
- Contém testes que cobrem 97% do código.
- Retorna os resultados em estrutura *AST* usando no padrão *Babel AST Format* (baseado no *ESTree*)
- Licença *MIT*.

2.6.1.5 *Benchmarks*

A tabela 2.1 representa um teste de desempenho na análise de diferentes bibliotecas que foi realizado recorrendo ao uso da biblioteca *BenchmarkJS* v2.1.3 e *Node.js* v6.9.5 *LTS* usando um *Macbook Pro Early* 2015 com as seguintes caraterísticas:

- Processador 2,7 GHz Intel Core i5
- Memória 8 GB 1897 Mhz DDR3

O código-fonte usado no teste pode ser consultado no anexo A.

Podemos inferir desta tabela que o *Esprima* e o *Acorn* foram os que tiveram o tempo de execução médio mais rápido.

Tabela 2.1: Tempos de execução usando *BenchmarkJS*

Biblioteca	**Esprima 3.1.3**	**Acorn 4.0.11**	**Espree 3.4.0**	**babylon 6.15.0**
jQuery.Mobile 1.4.5	61.3 ±17.5%	47.9 ±24.75%	114.2 ±4.3%	174.7 ±24.5%
Angular 1.6.1	80.5 ±4.4%	71.4 ±3.3%	176.6 ±25.0%	202.9 ±14.8%
React 15.4.2	11.7 ±9.5%	9.5 ±19.2%	28.0 ±32.1%	36.1 ±34.4%
Total	153.5 ms	128.8 ms	318.8 ms	413.7 ms

2.6.1.6 Resumo

Os quatro *parsers* analisados neste capítulo suportam todas as caraterísticas essenciais para o uso no projeto. Segundo o teste cujos resultados podem ser observados na tabela 2.1, o *Acorn* e o *Esprima* foram em média os mais rápidos na análise de diferentes bibliotecas JS.

Capítulo 3

Conceção e implementação

Neste capítulo é feita uma descrição detalhada da solução desenvolvida, designada por ASUND, onde o utilizador tem ao seu dispor vários módulos que permitem explorar e fazer *download* automático de repositórios JavaScript, análise estática dos repositórios para criação de uma base de dados de conhecimento e, por último, a classificação de aplicações ou código-fonte submetido a análise. A aplicação ASUND foi concebida para ser modular e funcionar sobre a plataforma *Node.js*. Uma informação mais detalhada está descrita nas secções seguintes.

3.1 Objetivos e requisitos

No sentido de construir uma solução que consiga atingir o objetivo principal, que passa por detetar o contexto e as bibliotecas usadas por uma aplicação submetida a avaliação, um conjunto de requisitos têm de ser cumpridos.

A solução terá que ser capaz de identificar o contexto de uma aplicação submetida a avaliação, detetando-o diretamente via indicadores específicos presentes no seu código-fonte, ou inferir através do contexto das bibliotecas em uso.

A deteção das bibliotecas poderá ser feita através de *metadata*, *imports* ou uso de *APIs*. Assim, terá de ser desenvolvida a capacidade de leitura de ficheiros de meta-dados, como, por exemplo, o ficheiro package.json, presente em módulos como o *npm*, suporte a identificação de *imports* introduzidos pela especificação *ES6* com o conceito nativo de módulos, e especificações não nativas como *AMD* e *CommonJS Module Spec* que foram implementadas respetivamente pela biblioteca *RequireJS* e plataforma *Node.js*.

Também deverá ser possível recolher estatísticas sobre os projetos submetidos como o uso das *API* e a precisão e sensibilidade do sistema.

É importante que a solução funcione isolada e seja facilmente adaptável a outro projeto, seja em forma de módulo ou *plugin*. A preferência será construir sobre a plataforma *NodeJS*, armazenando os dados numa base de dados externa.

3.2 Funcionalidades Gerais

Para cumprir os objetivos foram levantadas as diferentes funcionalidades listadas de seguida.

Command-Line Interface (*CLI*)

Arranque através de comandos com configuração rápida dos diferentes módulos.

Configurações

Definir ficheiros em formato *JSON* para configuração dos módulos.

Listar repositórios

Conectar com o *GitHub* e listar potenciais repositórios a analisar.

Download repositórios

Download dos repositórios e armazenamento apenas dos ficheiros relevantes.

Filtro de ficheiros

Filtrar ficheiros e pastas para análise estática.

Armazenar dados

Armazenar dados numa base de dados externa.

Deteção de módulos

Distinguir se o código-fonte define um *script* ou módulo.

Parse

Análise estática do código fonte com formação de AST.

Extração de *APIs*

Navegar na *AST* para proceder à extração dos nomes dos módulos e funções expostas a uso externo.

Extração de tópicos

Detetar o contexto do código navegando a AST na procura de caraterísticas próprias de cada tópico ou via inferência das bibliotecas usadas.

Deteção de bibliotecas

Leitura de ficheiros com meta-dados e *imports* via análise da AST ou uso de funções e objetos externos.

Geração de estatísticas

Geração de diferentes estatísticas de avaliação dos projetos e sistema. Cálculo da precisão e sensibilidade do sistema usando para tal um ficheiro de teste manual na raiz do projeto. Cálculo automático da precisão entre os *imports* detetados e os inferidos. Cálculo do uso da *API* extraída das bibliotecas.

3.3 Arquitetura

A arquitetura da solução ASUND consiste numa aplicação para *NodeJS* dividida em 3 módulos principais e um pacote de utilidades com funcionalidades partilhadas pelos módulos. Como ilustrado na figura 3.1 todos os módulos têm uma ligação a uma base de dados *NoSQL*, que poderá ser interna ou externa ao servidor, e a um espaço de armazenamento, onde serão guardados os repositórios descarregados para análise estática e onde o utilizador poderá colocar projetos ou aplicações para análise. Todos os módulos terão à sua disposição módulos e *API* externas para a realização das suas tarefas.

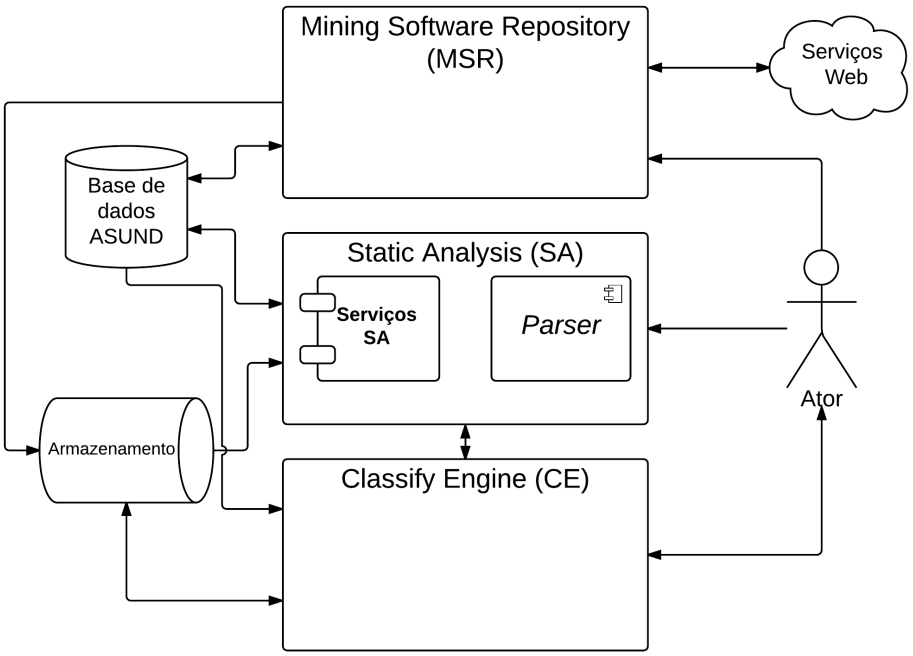

Figura 3.1: Arquitetura ASUND

Uma descrição mais detalhada dos módulos é efetuada nas secções seguintes.

3.4 Módulos

Foram criados três módulos principais designados por *Mining Software Repository* (MSR), *Static Analysis* (SA) e *Classify Engine* (CE). As suas responsabilidades são, respetivamente, explorar repositórios, fazer a análise estática de ficheiros e proceder à identificação e classificação.

Abordagens, problemas encontrados, decisões importantes e tecnologias usadas para cada módulo são detalhadas nas próximas secções.

3.4.1 Módulo *Mining Software Repository*

O módulo MSR (*Mining Software Repository*) é responsável por procurar no *GitHub* repositórios JavaScript que contenham *frameworks* ou bibliotecas.

Listar repositórios

A partir da *API* v3 do *GitHub*, mais especificamente o *search*, recolhe-se uma lista de repositórios a avaliar na seguinte rota: `"GET/search/repositories"`. Esta rota aceita os parâmetros detalhados na tabela 3.1.

Tabela 3.1: Detalhes dos parâmetros para pesquisa GitHub

Nome	**Tipo**	**Detalhe**
q	*string*	Palavras para pesquisa ou quantificadores
sort	*string*	Ordenação dos campos que poderá ser por *stars*, *forks* e ou *updated*
order	*string*	A ordenação ascendente (asc) ou descendente para os campos fornecidos
page	*number*	Em casa de paginação, devolve o número da página de resultado

De forma a tentar limitar o número de repositórios a serem devolvidos por esta pesquisa, foi decidido usar o número de estrelas (*stars*), para que sejam devolvidos apenas os repositórios mais populares. Este filtro será usado como potencial valor do uso e popularidade de uma biblioteca na comunidade.

Na listagem 3.1 está representado um exemplo da construção de um parâmetro q da tabela 3.1 que é necessário para produzir os resultados.

```
1  const query = `language:${language}+stars:">${minStars}"&sort=stars&order=desc`;
```

Listagem 3.1: Exemplo de uma *query* à *API* do github

Os valores variáveis estão detalhados na tabela 3.2. De notar que no campo *stars* o carácter maior (>) existe para limitar os resultados apenas a valores superiores.

A resposta fornecida pode ser incompleta visto que o GitHub limita-a a 100 resultados por página. Deste modo, assim para se obter uma lista completa é necessário acrescentar o campo *page*. Uma abordagem para obter a próxima página foi ler o campo *Link* presente no *header* da resposta, e através da aplicação da seguinte expressão regular `/<([^>]*)>;\s*rel="([\w]*)\"/g`,

Tabela 3.2: *Query* configurável

Nome	**Tipo**	**Detalhe**	**Valor Padrão**
language	*string*	Linguagem em uso	*javascript*
stars	*number*	Número de estrelas mínimas	100
sort	*string*	Ordenação do resultado pelo campo	*stars*
order	*string*	Sentido de ordenação para campo fornecidos	*desc*

obtermos a rota para a próxima página. A listagem 3.2 apresenta um exemplo simplificado com apenas os dados necessários à solução.

```
{
  "total_count": 19946,
  "incomplete_results": false,
  "items": [
    {
      "name": "react",
      "url": "https://api.github.com/repos/facebook/react",
      "language": "JavaScript",
      "default_branch": "master",
     }
   ]
}
```

Listagem 3.2: Resposta simplificada da rota *Search* da *API* do GitHub

Um dos problemas encontrados foi a limitação do número de pedidos por minuto que se pode realizar à *API*. No *Header* do pedido, é necessário introduzir o *User Agent* que contém limitações variáveis, na medida que um utilizador registado necessita de esperar um minuto para que o sistema atualize o número de pedidos permitidos.

Descarregar repositórios

Depois de obter a lista é necessário descarregar o repositório. Para tal, tomou-se a decisão de usar uma das funcionalidades do *GitHub* que permite descarregar um ficheiro *ZIP* de um *branch*. O endereço é a junção dos campos *url* e *default_branch* que foram armazenados previamente a partir da resposta 3.2. Após o processo estar completo é usada a biblioteca *Decompress* [Må17] listada no NPM para extrair o conteúdo do ficheiro *ZIP*. Apenas os ficheiros *Markdown* e JS foram extraídos pois são os únicos que têm interesse para a solução. Para minimizar o espaço ocupado no final da operação o ficheiro *ZIP* é eliminado.

De forma a manter a base de dados atualizada, há a possibilidade de se desenvolver, no futuro, um controlo de versões publicadas via *hash*. Esta ação permitirá descarregar, para nova análise, repositórios que tenham sofrido potenciais alterações ou novas adições, permitindo que o sistema se mantenha atualizado.

3.4.2 Módulo *Static Analysis*

O módulo SA (*Static Analysis*) será responsável por proceder à análise estática do código fonte.

Esta análise é feita com o suporte de duas ferramentas essenciais: *Acorn* e *AST-Types*.

O *Acorn* foi o *parser* eleito para fazer a transformação de código-fonte numa AST por ser melhor que os seus concorrentes em termos de velocidade, manter-se atualizado e ter a possibilidade de desenvolver *plugins* caso seja necessário desenvolver alguma funcionalidade extra.

Para proporcionar o suporte de especificações e bibliotecas mais recentes foi considerado o uso de dois *plugins*, o *Acorn-Jsx* que dá suporte à sintaxe *JSX* e o *Acorn-Object-Spread* que permite o uso da sintaxe *object spread*, ou seja, permite uma expressão ser expandida em locais onde múltiplos argumentos (por chamadas de função) ou múltiplos elementos (por *arrays* literais) são esperados. Ambas as funcionalidades não são suportadas até à data pelo *Acorn*.

Configuração *Acorn*

Este módulo foi configurado usando os parâmetros detalhados na tabela 3.3

Tabela 3.3: Configurações Acorn

Parâmetro	**Detalhe**	**Valor**
sourceType	Distingue módulo de *script*	*module* ou *script*
plugins	*Plugins* a usar em conjunto	*objectSpread* e *jsx*
ecmaVersion	Versão do *ECMAScript*	2017
allowReserved	Permite o uso de palavras reservadas	*true*
allowHashBang	Permite o uso de *hashbang*	*true*

3.4.2.1 Distinguir *Module* e *Script*

O JS antes da especificação 6 não tinha suporte nativo de módulos. Para colmatar a necessidade da criação de aplicações modulares, a comunidade desenvolveu soluções alternativas como o *CommonJS Modules* e o *AMD*. No passado, uma aplicação era apenas desenvolvida com uma sintaxe em forma de *script* semelhante à descrita na figura 3.2. O sucesso de bibliotecas que permitem criar aplicações modulares foi grande. Daí que o grupo de trabalho responsável por desenvolver novas especificações tenha decidido construir um suporte nativo com a publicação do *EcmaScript* 6. Este foi desenvolvido com o objetivo de agradar aos utilizadores de ambas as soluções não nativas. Por isso, foi criada uma nova sintaxe, ainda mais compacta, com as palavras-chave, *export* e *import*. Permite-se assim a criação de código mais modular que pode ser carregado assincronamente e que dá a possibilidade aos programadores de poderem criar um código fonte mais organizado usando apenas soluções nativas.

Como se pode observar na figura 3.2, o facto de as duas abordagem terem declarações tão distintas, levantou alguns problemas na análise. Uma solução encontrada para esta situação foi explicitar nas configurações se o código-fonte a analisar era um módulo ou um *script*. Assim,

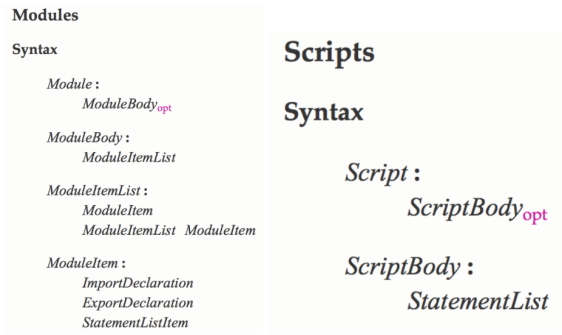

Figura 3.2: Exemplo da estrutura de um módulo vs *script*

um dos problemas encontrado no desenvolvimento deste módulo foi fazer esta distinção de forma automática, pois esta é necessária para evitar erros de análise sintática ou exceções.

```
// Test Script
  parser.parse('answer = 42', { sourceType: 'script' });
// Result
  Program {
  type: 'Program',
  body: [ ExpressionStatement {
  type: 'ExpressionStatement', expression: [Object] } ],
  sourceType: 'script'
  }
```

```
// Test Module
  parser.parse('import { sqrt } from "math.js"', { sourceType: 'module' });
// Result
  Program {
  type: 'Program',
  body:
  [ ImportDeclaration {
  type: 'ImportDeclaration', specifiers: [Object], source: [Object] } ],
sourceType: 'module'
  }
```

Figura 3.3: Exemplo do *parse* de um ficheiro do tipo *script* e do tipo *module*

Um exemplo do uso de ambos os tipos pode ser encontra na figura 3.3. Num *script* as palavras *import* ou *export* não são reservadas e por isso não devem ser usadas no contexto de importar e exportar objetos ou funções. No caso do módulo, a palavra *import* é detetada como o nó *ImportDeclaration* e a palavra *export* como o nó *ExportDeclaration*. Em caso de uso indevido, os analisadores estáticos como o *Acorn* invocarão uma exceção, pois teremos em mãos um erro de análise.

A listagem 3.3 apresenta o erro que teremos caso se faça análise de um módulo em modo *script*.

```
{
  // Module as Script
  parser.parse('import { sqrt } from "math.js"', { sourceType: 'script' });
  // Result
  Error: 'import' and 'export' may appear only with 'sourceType: module'
}
```

Listagem 3.3: Exemplo do *parse* de um ficheiro do tipo *module* com o tipo *script*

A solução que se implementou para conseguir uma forma automática de distinguir um módulo de um *script*, de modo a evitar erros no *Acorn* e melhorar também a futura análise por indicadores,

foi aplicar a cada ficheiro um teste da expressão regular ilustrada na listagem 3.4, onde se procura pelas palavras-chave que se usam para a definição de módulos nativos ou não.

```
1 {
2   /module\.exports|exports\.|export |import|define\(|require\(/
3 }
```

Listagem 3.4: Expressão regular para detetar se o código-fonte descreve um módulo

3.4.2.2 Análise da AST

Após a transformação do código-fonte numa AST procede-se à exploração das árvores. Para tal, foi usado um módulo com o nome *Ast-Types*, compatível com a especificação *ESTree*. Este é um explorador AST que permite definir que tipo de nó queremos analisar quando visitamos a árvore e que usa um *wrapper* sobre os mesmos nós, designados por *NodePath*, que dá acesso ao pai e também permite gerar um lista de variáveis presentes no seu *scope* [New17].

Tal informação será usada para extrair o nome de funções ou objetos expostos para encontrar chamadas externas e tópicos em uso.

Foram criados, para o efeito, 4 ficheiros com as definições necessárias para cada caso, ou seja, exposição de funções ou objetos, chamadas externas, *imports* e tópicos. Todas as especificações usadas atualmente foram suportadas desde o *ES5* e versões anteriores, que é suportado amplamente por todas as plataformas, até ao *ES2017*, que é a especificação mais recente. O suporte de módulos também está incluído, seja através da leitura dos *imports* e *exports* declarados em *ES6*, ou através do uso das bibliotecas de módulos não nativos.

3.4.3 Módulo *Classify Engine*

O módulo CE (*Classify Engine*) é o único que é dependente diretamente da funcionalidade de outro módulo. Este necessita de funcionalidades do módulo SA para a análise de aplicações submetidas. A sua dependência passa pela análise estática do código-fonte para que se consiga uma extração dos tópicos e *imports*, meta-dados ou chamadas exteriores ao sistema.

A classificação atribuída aos ficheiros em análise passa por expor o seu contexto, por deteção direta ou inferida, através das bibliotecas em uso que também podem ser detetadas diretamente ou inferidas.

```
1 {
2   "path",
3   "files": [],
4   "folders": [],
5   "data": {}
6 }
```

Listagem 3.5: Resultado da avaliação - Estrutura de uma pasta

O resultado que este módulo produz é um objeto do tipo *JSON* que descreve a estrutura do projeto em análise assim como os resultados para todos os ficheiros e pastas.

A listagem 3.5 é um exemplo de uma pasta e contém os diferentes campos:

- *path* - É o caminho da pasta.
- *files* - Contém todos os ficheiros contidos dentro da pasta.
- *folders* - Contém todas as pastas que são filhas.
- *data* - Contém todos os dados de análise, desde tópicos e bibliotecas encontradas, assim como estatísticas, a avaliação desta pasta e com os seguintes campos:
  - *test* - Contém o resultado da leitura do ficheiro *asund-manual-test.json*.
  - *topics* - Tópicos encontrados dentro da pasta com o número de ficheiros encontrados por tópico.
  - *libs* - Bibliotecas encontradas dentro da pasta com o número de ficheiros encontrados por biblioteca.
  - *stats* - Cálculo da precisão e sensibilidade usando os dados encontrados no campo *test*, *topics* e *libs*.

Através do campo *data* conseguimos um resumo do sistema ou de cada pasta. Este, contém o número de ficheiros encontrados tanto por tópico como por biblioteca. Caso se pretenda a validação do sistema é necessário colocar na raiz da pasta ou projeto a analisar, um ficheiro *JSON* com o nome *asund-manual-test* e com a estrutura ilustrada na listagem 4.2. Neste ficheiro encontram-se definidos três *arrays* com os tópicos e bibliotecas que foram manualmente testadas. A partir destes dados será possível calcular a precisão e a sensibilidade.

Na listagem de código 3.6 está representada a estrutura do resultado produzido depois da análise de um ficheiro. Os seus campos são:

- *path* - É o caminho para o ficheiro.
- *libs* - São todas as bibliotecas detetadas e distribuídas pelos seguintes campos:
  - *detected* - Lista de todas as bibliotecas detetadas diretamente por importações presentes no código.
  - *inference* - Mapeamento de todas as bibliotecas que foram inferidas através da deteção do uso da sua *API* com o número de diferentes indicadores encontrados.
  - *usage* - Percentagem de uso da *API* das bibliotecas inferidas.
- *topics* - São todos os tópicos detetados e distribuídos pelos campos:
  - *ecmascript* - A mais recente especificação JS encontrada.
  - *detected* - Tópicos encontrados diretamente.

- *inference* - Tópicos encontrados por inferência através do uso de bibliotecas. A sua distribuição é feita pelos seguintes campos:
  - *detectedLibs* - Tópicos encontrados por inferência através de bibliotecas detetadas.
  - *inferenceLibs* - Tópicos encontrados por inferência através de bibliotecas que foram inferidas.

```
{
  "path",
    "libs": {
    "detected": [],
    "inference": {},
    "usage": {}
  },
  "topics": {
    "detected": [],
    "inference": {
      "detectedLibs": {},
        "inferredLibs": {},
     }
  }
}
```

Listagem 3.6: Resultado da avaliação - Estrutura de um ficheiro

3.4.4 Fluxo de funcionamento e interação

Na figura 3.4, está ilustrado o fluxo simplificado do funcionamento dos módulos.

O módulo MSR, depois de iniciado com os parâmetros corretos, irá listar os repositórios a analisar e armazena os seus dados. Caso existam repositórios por analisar, estes são descarregados e descompactados para serem armazenados num espaço pré-determinado. Com estes dados e depois de inicializado, o módulo SA lista todos os repositórios prontos a analisar. Caso tenha sido descarregado com sucesso, o repositório será analisado estaticamente e indexado. Por fim, o módulo CE classificará um conjunto de projetos ou aplicações submetidas a avaliação. Para tal, identifica diretamente os tópicos e bibliotecas. Estas, também podem ser inferidas através do cruzamento dos dados previamente indexados com os identificados usando a análise estática.

Todos os módulos podem ser usados em modo independente com exceção do modulo CE, que necessita de algumas funcionalidade desenvolvidas no módulo SA. Tanto o módulo SA como o módulo CE necessitam de dados que terão de ser introduzidos numa base de dados de forma manual ou através de outro sistema compatível. É recomendado o uso dos três módulos para preenchimento da base de dados e proceder à classificação dos projetos.

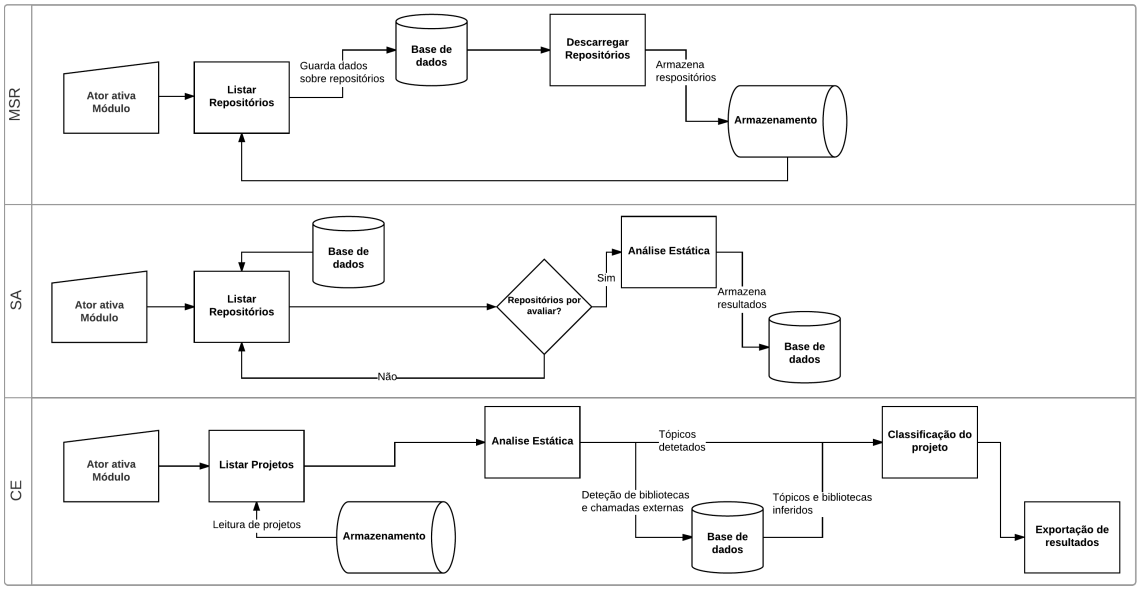

Figura 3.4: Fluxo simplificado do funcionamento dos módulos

3.5 Identificação de bibliotecas

A identificação das bibliotecas é feita através de 3 métodos: meta-dados, importações e uso da sua *API*.

O método mais fácil e direto é realizar a sua identificação através das importações das mesmas.

Em alguns projetos, são fornecido ficheiros com meta-dados. Por exemplo, num módulo NPM é obrigatória a existência de um ficheiro *JSON* com o nome *package* onde estão detalhados quais as bibliotecas presentes e potencialmente em uso por tal aplicação ou pacote. Um dos problemas é que estes dados destinam-se ao projeto todo, e não a pastas ou ficheiros individuais.

Quando o sistema de módulos não é usado, ou não está presente, o uso das *APIs* das bibliotecas torna-se necessário.

Esta *API* é recolhida através da análise dos ficheiros JS presentes no repositório. Nomes de funções e módulos são indexados numa base de dados que depois serão alvo de uma pesquisa binária usando para tal termos identificados no código a avaliar. Esses termos são chamadas a funções ou objetos externos.

De modo a reduzir os falsos positivos, pois diferentes bibliotecas poderão ter interfaces semelhantes com o uso de nomes iguais, são recolhidos, para além dos objetos e funções, as suas propriedades e argumentos.

Um dos problemas que se enfrentou na identificação de bibliotecas foi no desenvolvimento de uma heurística que permita filtrar componentes ou bibliotecas definidas internamente. Como a organização dos projetos, tal como os repositórios, sofre muitas vezes de uma falta de estrutura, torna-se árdua a tarefa de distinguir módulos e outros componentes de bibliotecas externas. Para manter uma sensibilidade próxima dos 100% não se filtra as pasta ou ficheiros submetidos a

análise. Porém, tal ação prejudicará a precisão em geral.

3.6 Tópicos

O contexto de uma aplicação submetida é definida pela atribuição de um conjunto de tópicos. Estes foram manualmente determinados em 5 categorias: *backend*, *frontend*, *desktop*, *mobile* e *canvasapp*.

A deteção destas categorias pode ser direta através da procura de indicadores ou por inferência, via uso de bibliotecas que foram desenvolvidas para determinados contextos.

Uma descrição destes tópicos pode ser encontrado nas secções que se seguem.

De notar que é também detetado um tópico especial, que não define o contexto mas a especificação usada. Este tópico tem como nome *ecmascript* e tem como valor padrão, o valor 5. Caso seja encontrado algum indicador de que uma especificação mais recente foi usada, esse valor é atualizado. Ficará sempre o número da especificação mais recente.

3.6.1 *Backend*

Este contexto determina se um módulo ou *script* foi concebido para funcionar dentro de um servidor ou ambiente semelhante.

Nesta solução apenas se procura pelo uso dos módulos desenvolvidos para o core do *Runtime JavaScript Node.js* que é a solução para criar aplicações de servidor mais popular no contexto de desenvolvimento em JS.

Como o *Node.js* usa um modelo de *imports* derivado do *CommonJS* a solução irá detetar que módulos em uso podem pertencer ao *Node.js*.

3.6.2 *Frontend*

Definiu-se como *frontend*, todo o código JS responsável por criar e manipular a interface de uma aplicação. Uma interface é geralmente definida através de HTML e CSS, que, depois de interpretada por uma plataforma, como por exemplo um navegador, é transformado numa árvore DOM. O acesso à manipulação dessa árvore é feita através da interface *Document*. Outras funcionalidades poderão ser fornecidas para a criação de aplicações mais complexas e ricas através da interface *Window*.

Para detetar este tópico procura-se no código chamadas às *API* das interfaces *Windows* e *Document* que são geralmente fornecidos como objetos JS. As suas propriedades são definidas pelo *DOM Core API* que são recomendações da especificação DOM da W3C[W3C17].

3.6.3 *Desktop* e *Mobile*

Com o aumento do poder do motor JS V8 e *Node.js*, diferentes *frameworks* foram criadas para que aplicações desenvolvidas em JS com o mesmo código possam funcionar de modo idêntico em multiplataforma como computadores de secretária, portáteis, *tablets* e *smarphones*.

O *desktop* é atribuído a aplicações desenhadas para funcionar em sistemas operativos como Windows, *MacOS* e Linux entre outros, que são geralmente concebidos para funcionar em computadores pessoais ou portáteis.

O tópico *mobile* é aplicado para sistemas operativos como *iOS*, *Android* ou *Windows Phone*, sistemas operativos mais otimizados para dispositivos móveis e geralmente com ecrãs sensíveis ao toque.

A deteção é efetuada pela procura de chamadas externas ou importação das *frameworks* ou bibliotecas desenhadas para ambos os contextos.

3.6.4 Canvasapp

Este tópico envolve todo o tipo de aplicações que fazem uso do elemento *canvas*. Este permite a criação de aplicações e jogos 2D ou 3D com gráficos simples ou complexos, usando tanto a procura pelo uso do elemento como o uso da sua propriedade *getContext* para a sua deteção.

Capítulo 4

Execução e validação

Este capítulo apresenta os resultados obtidos e como se procedeu à execução dos módulos desenvolvidos. De forma a validar o sistema, serão testados vários projetos escolhidos aleatoriamente para cada um dos tópicos com e sem *imports* declarados.

4.1 Execução

Para executar a solução ASUND e proceder aos testes, será necessário executar os 3 módulos pela seguinte ordem: MSR, SA e CE.

Para executar a solução, as seguintes tecnologias têm de estar previamente instaladas:

- *Node.js* com suporte a EcmaScript 2017 (versão 7.6 ou superior).
- *MongoDB* 3.4 ou superior.
- *Package Manager* (NPM ou Yarn)

Se todas as tecnologias estiverem devidamente instaladas seguem-se os seguintes passos:

1. Instalar todas as dependências necessárias usando o *Package Manager*.
2. Colocar uma instância da base de dados *MongoDB* a correr.
3. Executar os módulos por ordem.

```
1 NODE_ENV=x node main.js --module m --repo_path y --eval_folder z  --stats b
```

Listagem 4.1: Comando de execução da solução

Foram desenvolvidos comandos *CLI* para executar e configurar os módulos. Para tal na raiz da solução deve-se executar o comando da listagem 4.1 com os seus parâmetros detalhados na tabela 4.1.

Tabela 4.1: Detalhes dos parâmetros para o comando de arranque da solução

Parâmetro	**Detalhes**
`NODE_ENV`	Variável de ambiente que poderá ter os valores *production*, *stage* e *development*. É responsável por carregar as configurações especiais de cada ficheiro definido com o mesmo nome.
`module`	Parâmetro que define que módulo será invocado. Poderá ter os valores *msr*, *sa* e *ce*. O valor por omissão será *ce*. (opcional)
`repo_path`	Caminho para a pasta onde se armazena ou estão os repositórios a analisar. (opcional)
`eval_folder`	Caminho para a pasta com os projetos a serem classificados. (opcional)
`stats`	true para produzir estatísticas sobre o sistema. (opcional)

Após executar tanto os módulos MSR e SA, a base de dados de conhecimento deverá estar completa. O módulo CE poderá agora ser executado sobre uma pasta contendo projetos a avaliar.

Para produzir um conjunto de estatísticas sobre os projetos avaliados de forma a validar o sistema, um conjunto de ficheiros com o nome *asund-manual-test* deve ser colocado na raiz do projeto ou pasta que queremos avaliar. A listagem 4.2, mostra uma lista de tópicos que definem o contexto da pasta em questão e todos os módulos e bibliotecas internas e externas em uso. O campo *ex* indica as bibliotecas externas.

```
{
  "topics" : {
      "ecmascript": 5,
      "found": ["backend"]
  },
  "libs" : {
      "all": ["react", "react-dom", "electron", "internal"],
      "ex": ["react", "react-dom", "electron"]
  }
}
```

Listagem 4.2: Exemplo do conteúdo do ficheiro *asund-manual-test*

4.2 Validação

Usando os dados de teste que servirão como um classificador externo de confiança e os tópicos e as bibliotecas encontrados pelo classificador da solução ASUND, será possível calcular a precisão e a sensibilidade do sistema.

A precisão é calculada através da fórmula $P = X/Y$, onde P é a precisão e X são os tópicos ou bibliotecas fornecidos pelos testes manuais (fonte de confiança) e que foram encontrados pelo sistema, ou seja, apenas os encontrados no conjunto dos expectáveis. Y são todos os itens fornecidos que foram encontrados pelo sistema.

A sensibilidade é calculada usando a fórmula $S = Z/R$, onde S é a sensibilidade, Z são todos os elementos fornecidos pelos testes manuais que foram encontrados pelo sistema e R são todos os os elementos fornecidos pelos testes manuais.

4.3 Testes

Para testar o sistema, foram selecionados 5 projetos ou amostras retiradas a partir do *GitHub*. Estes são diversos e abrangem os diferentes contextos e uso de bibliotecas que a solução permite detetar.

Todos os teste foram realizados usando as seguintes tecnologias instaladas:

- *Node.js* v7.7.1
- *NPM* 4.5.0
- *MongoDB* 3.4

Nas próximas secções estão detalhados todos os projetos e a sua análise manual que foram usados na validação do sistema. Os testes manuais podem ser encontrados na secção do anexo B.

Projeto 1

Este projeto de nome *electron-with-create-react-app* é um exemplo criado para o sítio *freecodecamp.com*, onde se cria uma aplicação *desktop* usando a biblioteca *Electron* e *React*.

Projeto 2

O segundo projeto é um módulo chamado *node-graceful-fs* para a plataforma *Node.js*, que permite melhorar com várias funcionalidades a *API fs* do *Node.js*.

Projeto 3

HiApp é uma aplicação híbrida para *iOS* e *Android* desenvolvida usando as *framework Framework7* e *VueJS*.

Projeto 4

Reactive-native-start foi o quarto projeto selecionado. Tal como o projeto 3 é uma aplicação híbrida para *iOS* e *Android* mas desenvolvida usando as bibliotecas *React-native* e *React*.

Projeto 5

O projeto 5, designado *simple_canvas_game*, é um projeto não modular que usa o elemento *canvas* para criar um jogo simples. Este não faz uso de qualquer biblioteca externa ou interna.

4.4 Resultados

Foram executados testes sobre 5 projetos publicados no *GitHub* com diferentes contextos, e uso de diferentes *frameworks* e bibliotecas. Os resultados foram divididos em 4 partes: a avaliação do sistema para a deteção automática, inferência através do uso das bibliotecas encontradas tanto via *imports* encontrados ou avaliação da *API*, o somatório dos dois e por fim avaliação da deteção direta dos tópicos com a inferência apenas da deteção direta de bibliotecas.

Na figura 4.1 está ilustrado um gráfico que contém a avaliação do sistema filtrado pelos dados referentes à deteção direta tanto dos tópicos como das bibliotecas.

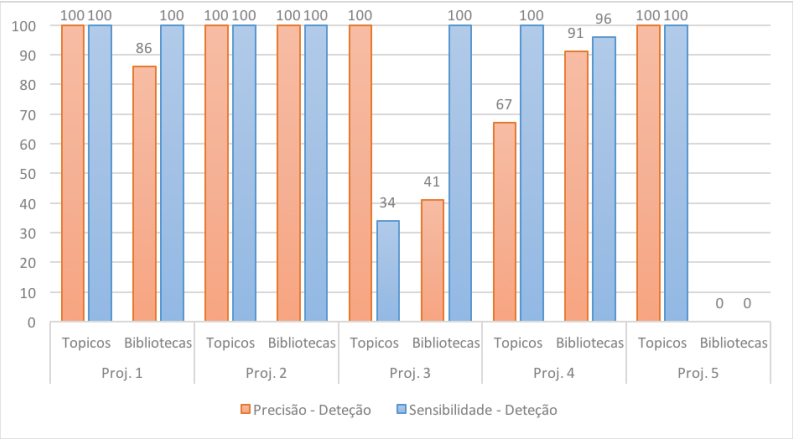

Figura 4.1: Avaliação do sistema
- Filtrado por deteção

Para a obtenção da 4.2 foi aplicado um filtro que limita os dados apenas à inferência tanto dos tópicos como das bibliotecas. De notar que se decidiu incluir na deteção dos tópicos a inferência das bibliotecas tanto através de deteção direta das bibliotecas como as inferidas pela *API*.

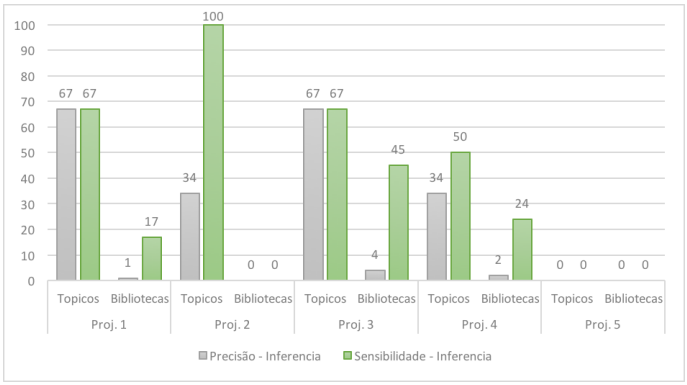

Figura 4.2: Avaliação do sistema
- Filtrado por inferência

A figura 4.3 ilustra a capacidade do sistema como um todo, mostrando os resultados obtidos tanto pela deteção como pela inferência.

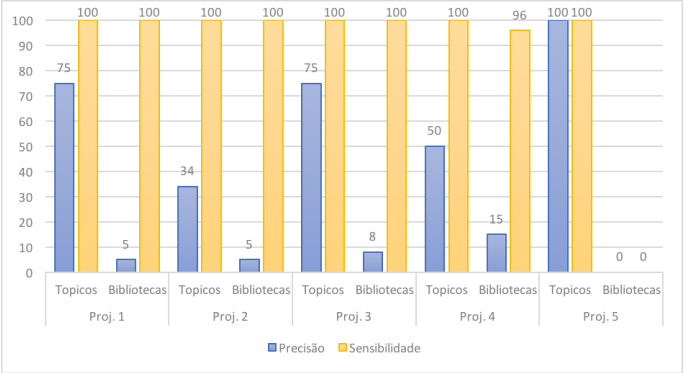

Figura 4.3: Avaliação do sistema

Por último na figura 4.4 filtraram-se os resultados pelos tópicos onde apenas está incluído a deteção direta dos tópicos e os inferidos através da deteção direta das bibliotecas.

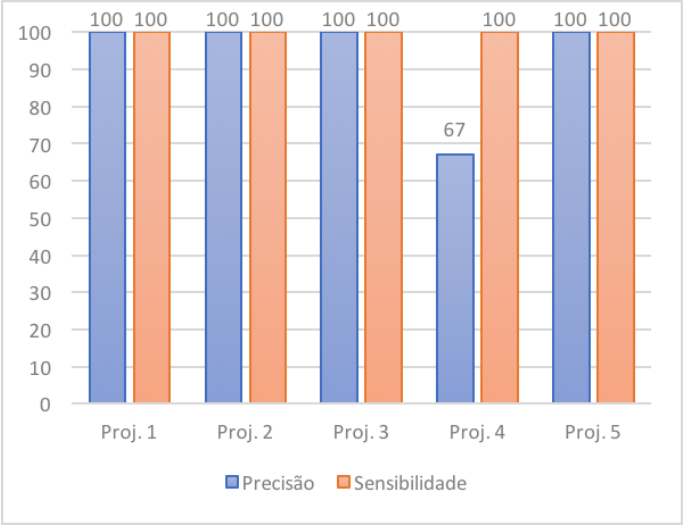

Figura 4.4: Avaliação do sistema - Filtrado por deteção e importações

4.5 Discussão de resultados

Filtrando os resultados por deteção direta, como podemos observar na figura 4.1, poderíamos concluir que este tipo de deteção poderá ser suficiente para produzir uma classificação em parte dos projetos. Mas tal como acontece nos projetos 3 e 4, poderá haver outros projetos que usam bibliotecas externas para executar funcionalidades em certos contextos e por isso existiram falhas na classificação usando apenas este método. Podemos concluir, que a deteção direta dos tópicos não é suficiente. Para inferir os tópicos em falta será necessário proceder à deteção das bibliotecas

em uso. No projeto 3 a precisão por deteção direta das bibliotecas baixou visto que este contém dois ficheiros que são usados para as diferentes plataformas *mobile*. O código-fonte deste ficheiros importam uma pasta contendo a lógica da aplicação que será interpretada pelo sistema com um módulo ou biblioteca. Esta situação poderá acontecer porque caso uma pasta tenha um ficheiro de nome *index.js* a solução irá detetar tal pasta como um módulo que, como neste caso, não é.

Na identificação dos tópicos para projetos que usem uma solução modular, se for incluída a deteção direta e a inferência a partir de bibliotecas detetadas conseguimos melhorar os resultados na generalidade, tal como podemos observar na figura 4.4. Ainda assim, poderá existir uma anomalia devido à partilha de nomes entre diferentes bibliotecas. Apesar de alguns projetos seguirem normas definidas pela comunidade, grande parte seguem estruturas próprias. Levanta-se assim, um problema na identificação, pois estas poderão ser bibliotecas externas, internas, públicas ou privadas. Não foi encontrada uma solução viável, sendo por isso decidido que todos os ficheiros poderão ser módulos ou bibliotecas. A consequência poderá ser uma descida na precisão pois diferentes bibliotecas com o mesmo nome podem ter contextos diferentes.

Uma boa sensibilidade e precisão do contexto da aplicação depende assim da deteção das bibliotecas em uso, o que aumenta a importância da sua deteção. A deteção de bibliotecas na presença de meta-dados e importações é, como observamos na figura 4.4 e 4.1, um benefício para a solução, mas estes dados nem sempre estão disponíveis. Em código-fonte do tipo *script*, sem uso de bibliotecas de módulos não nativos, a tendência é usar bibliotecas já carregadas no *scope* global. É assim necessário para esses casos usar outro tipo de indicadores, como por exemplo, o uso das *APIs* dessas bibliotecas.

Da figura 4.2, observamos que tanto a precisão como a sensibilidade na deteção de bibliotecas a partir da *API* não contribui para a melhoria do resultado final.

Os problemas que podem contribuir para o aumento do ruído na base de conhecimento são:

- A dificuldade de identificar se um repositório é uma biblioteca, uma *framework*, um simples tutorial ou outro tipo de projeto, faz com que se analise todos os repositórios.
- Os repositórios que contêm uma estrutura própria, dificultam o filtro de ficheiros ou pastas que não se deve analisar, pois poderão criar duplicações da *API*.
- Bibliotecas, incluindo também as que seguem uma estrutura de pasta comum, que injetam outras bibliotecas no *scope* global.
- Bibliotecas privadas ou públicas com o mesmo nome.

É de notar que o segundo e o quarto ponto também podem acontecer na extração de invocações às *APIs* de bibliotecas externas. Por exemplo, a solução quando avalia o repositório *Node.js* extrai como *API* um conjunto de módulos que têm nomes semelhantes a outros repositórios. Isso também aconteceu com bibliotecas populares como o *jQuery*. Se um projeto fizer uso destes módulos ou *APIs*, irá provocar um incremento de falsos positivos ao inferir via importações ou uso da *API*, devido à adição ao resultado final de inúmeras bibliotecas em que os seus repositórios foram mal analisados pelos diferentes motivos descritos nos pontos anteriores.

Todos estes problemas, de difícil resolução, contribuem para o aumento do número de falsos positivos por cada ficheiro analisado. Estes erros, irão depois acumular na análise de uma pasta, o que irá influenciar bastante a precisão e a sensibilidade do sistema na análise de um projeto.

Pode-se assim concluir, com o suporte das figuras 4.4 e 4.3 que ilustra uma visão geral dos resultados, que se forem ignorados os resultados da deteção das bibliotecas via inferência pela *API*, a solução tem uma precisão e sensibilidade ótima para avaliação do contexto.

Capítulo 5

Conclusões

Nesta capítulo é analisada a satisfação dos objetivos propostos e é descrito o trabalho futuro a realizar.

5.1 Satisfação dos objetivos

Durante esta dissertação foi estudada e desenvolvida uma solução capaz de classificar aplicações JavaScript. Foi idealizado um conjunto de 3 módulos que através da análise estática do código-fonte recolhem um conjunto de indicadores de repositórios e aplicações com o intuito de identificar tópicos, importações e exportações de módulos, e extrair a *API*.

Usando uma abordagem semelhante a um sistema RI, a solução explora repositórios para detetar diretamente tópicos e extrair a sua *API*. À posteriori a classificação de aplicações ou projetos submetidos a avaliação é realizada através da deteção direta dos tópicos ou inferindo através do uso de bibliotecas.

A deteção das bibliotecas é feita da através de *metadata*, *imports* quando disponível, ou ainda pela identificação do uso de *APIs* específicas de cada biblioteca.

Foi também desenvolvido um sistema para recolher estatísticas sobre os projetos submetidos como por exemplo o uso das *API*, a precisão e a sensibilidade do sistema e a distribuição dos tópicos e uso das bibliotecas por pasta.

Dos resultados, concluímos que a deteção direta dos tópicos, juntamente com os inferidos a partir da análise de *metadata* e *imports* são suficientes para classificar os projetos e seus conteúdos com boa precisão, quando estes são desenvolvidos usando sistemas de módulos. Na omissão de meta-dados e importações, inferir o uso de bibliotecas através da *API* não se mostrou como uma solução com boa sensibilidade e precisão, devido aos diversos problemas por resolver para melhorar a prevenção da poluição da base de conhecimentos.

A solução funciona tanto de forma isolada, exportando o resultados em ficheiros *JSON*, como pode ser facilmente adaptada a outro projeto, seja em forma de módulo como *plugin*.

5.2 Evolução futura

Analisado o trabalho desenvolvido e os resultados obtidos, retiraram-se algumas conclusões sobre o que deve ser realizado no futuro com vista a melhorar a qualidade dos resultados, principalmente na parte de inferir o uso de bibliotecas a partir da *APIs* usada.

Em primeiro lugar, seria útil desenvolver a capacidade do módulo MSR de detetar possíveis mudanças nas bibliotecas para que a base de conhecimento esteja o mais atualizada possível ou consiga detetar que versão poderá estar em uso. Uma abordagem possível, seria usar o controlo de versões dos repositórios para detetar possíveis mudanças na interface das bibliotecas ou módulos.

De forma a melhorar a precisão da deteção das bibliotecas via *API*, funcionalidade importante quando o código-fonte não contém *imports* descriminados, terão de existir estudos mais aprofundados para resolver diversos problemas que surgiram onde a abordagem atual provou a existência de pouca sensibilidade e precisão. Será necessário desenvolver um filtro capaz de ignorar repositórios que não são *frameworks*, bibliotecas ou módulos. Para encontrar indicadores que os relevem, a leitura dos tópicos, funcionalidade em implementação pelo *GitHub*, ou leitura dos ficheiros *Markdown*, poderão ser possíveis abordagens. Também há necessidade de um estudo mais profundo na deteção do tipo de pasta ou ficheiros. Terá de ser possível destingir ficheiros ou pastas de teste e ficheiros compilados ou com outras bibliotecas, para que estes sejam ignoradas. O filtro atual necessita de ser melhorado manualmente ou por aplicações de aprendizagem automática pois este funciona apenas para alguns padrões gerais de estruturação dos repositórios ou projetos.

Devido à comunidade de JS criar *compile-to-js languages*, terá de ser desenvolvido suporte para os mesmo, pois o aumento da popularidade do seu uso poderá provocar uma diminuição da sensibilidade do sistema.

Os resultados de um grupo de avaliações poderão no futuro ser usados como dados de entrada para a construção de uma máquina de aprendizagem automática.

Referências

[Air17] AirBnb. Airbnb javascript style guide. Disponível em https://github.com/airbnb/javascript, acedido a última vez em 9 de junho de 2017, 2017.

[Alh10] Anwar A Alhenshiri. Web Information Retrieval and Search Engines Techniques. *World Wide Web Internet And Web Information Systems*, pages 55–92, 2010.

[Ali17] Elbert Alias. wappalyzer. Disponível em https://wappalyzer.com, acedido a última vez em 28 de Janeiro de 2017, 2017.

[App17] Apple. Javascriptcore. Disponível em https://trac.webkit.org/wiki/JavaScriptCore, acedido a última vez em 17 de Janeiro de 2017, 2017.

[bui17] buildwith. About. Disponível em https://builtwith.com/about, acedido a última vez em 28 de Janeiro de 2017, 2017.

[CLC16] Valerio Cosentino, Javier Luis e Jordi Cabot. Findings from GitHub: Methods, Datasets and Limitations. *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 137–141, 2016. URL: http://dl.acm.org/citation.cfm?doid=2901739.2901776, doi:10.1145/2901739.2901776.

[Dev17] Google Developers. Chrome v8. Disponível em https://developers.google.com/v8/, acedido a última vez em 17 de Janeiro de 2017, 2017.

[ESL17] ESLint. An esprima-compatible javascript parser. Disponível em https://github.com/eslint/espree, acedido a última vez em 28 de Janeiro de 2017, 2017.

[EST17] ESTree. Estree. Disponível em https://github.com/estree/estree, acedido a última vez em 2 de fevereiro de 2017, 2017.

[Fre17] FreeCodeCamp.com. Freecodecamp. Disponível em https://github.com/freeCodeCamp/freeCodeCamp, acedido a última vez em 9 de junho de 2017, 2017.

[Git17a] Github. Electron documentation. Disponível em http://electron.atom.io/docs/, acedido a última vez em 20 de Janeiro de 2017, 2017.

[Git17b] Github. Folder structure conventions. Disponível em https://github.com/kriasoft/Folder-Structure-Conventions, acedido a última vez em 2 de fevereiro de 2017, 2017.

[Git17c] Github. Github about. Disponível em https://github.com/about, acedido a última vez em 25 de Janeiro de 2017, 2017.

[Git17d] Github. Language trends 2016. Disponível em https://github.com/blog/2047-language-trends-on-github, acedido a última vez em 15 de Janeiro de 2017, 2017.

[Goo17] Google. Google code. Disponível em https://code.google.com, acedido a última vez em 25 de Janeiro de 2017, 2017.

[Har13] Jeff Harrell. Node.js at paypal. Disponível em https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/, acedido a última vez em 15 de Janeiro de 2017, 2013.

[Has08] Ahmed E Hassan. The road ahead for Mining Software Repositories. *2008 Frontiers of Software Maintenance*, pages 48–57, 2008. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4659248, doi:10.1109/FOSM.2008.4659248.

[Hav17] Marijn Haverbeke. A small, fast, javascript-based javascript parser. Disponível em https://github.com/ternjs/acorn, acedido a última vez em 28 de Janeiro de 2017, 2017.

[Hid17] Ariya Hidayat. Ecmascript parsing infrastructure for multipurpose analysis. Disponível em http://esprima.org, acedido a última vez em 28 de Janeiro de 2017, 2017.

[Int17a] ECMA International. Ecma-262. Disponível em http://www.ecma-international.org/publications/standards/Ecma-262.html, acedido a última vez em 23 de Junho de 2017, 2017.

[Int17b] ECMA International. Ecmascript 2018 language specification. Disponível em https://tc39.github.io/ecma262/, acedido a última vez em 23 de Junho de 2017, 2017.

[Len14] Lennovate. Ecmascript 2017 language specification. Disponível em http://mean.io, acedido a última vez em 17 de Janeiro de 2017, 2014.

[Med17] Slashdot Media. Sourceforge about. Disponível em https://sourceforge.net/about, acedido a última vez em 25 de Janeiro de 2017, 2017.

[Mic17] Microsoft. Codeplex help. Disponível em http://www.codeplex.com/site/help, acedido a última vez em 25 de Janeiro de 2017, 2017.

[MLF15] Magnus Madsen, Benjamin Livshits e Michael Fanning. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. *Ase*, pages 499–509, 2015. URL: http://doi.acm.org/10.1145/2491411.2491417, doi:10.1145/2491411.2491417.

[Må17] Kevin Mårtensson. Decompress. Disponível em https://github.com/kevva/decompress, acedido a última vez em 7 de junho de 2017, 2017.

[New17] Ben Newman. Ast types. Disponível em https://github.com/benjamn/ast-types, acedido a última vez em 7 de junho de 2017, 2017.

[Pad13] Senthil Padmanabhan. How we built ebay's first node.js application. Disponível em http://www.ebaytechblog.com/2013/05/17/how-we-built-ebays-first-node-js-application/, acedido a última vez em 15 de Janeiro de 2017, 2013.

[Pho17] Phonegap. Platform security. Disponível em https://github.com/phonegap/phonegap/wiki/Platform-Security, acedido a última vez em 20 de Janeiro de 2017, 2017.

[POW11] D.M.W. POWERS. Evaluation: From Precision, Recall and F-Measure To Roc, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011. URL: http://www.bioinfopublication.org/files/articles/2_1_1_JMLT.pdf, doi:10.1.1.214.9232.

[PSB11] Wouter Poncin, Alexander Serebrenik e Mark Van Den Brand. Process Mining Software Repositories. *2011 15th European Conference on Software Maintenance and Reengineering*, pages 5–14, 2011. doi:10.1109/CSMR.2011.5.

[Sim17] Kyle Simpson. You-dont-know-js. Disponível em https://github.com/getify/You-Dont-Know-JS, acedido a última vez em 9 de junho de 2017, 2017.

[SP13] Manish Sharma e Rahul Patel. A Survey on Information Retrieval Models, Techniques And Applications. *International Journal of Emerging Technology and Advanced Engineering*, 3(11):542–545, 2013.

[Spr17] Spring. Javascript package managers. Disponível em https://spring.io/understanding/javascript-package-managers, acedido a última vez em 25 de Janeiro de 2017, 2017.

[sta17] stackoverflow. Developer survey results 2016. Disponível em http://stackoverflow.com/research/developer-survey-2016#technology, acedido a última vez em 15 de Janeiro de 2017, 2017.

[Tec17] Unity Technologies. Editor. Disponível em https://unity3d.com/unity/editor, acedido a última vez em 16 de Janeiro de 2017, 2017.

[W3C17] W3C. Dom - living standard. Disponível em https://dom.spec.whatwg.org, acedido a última vez em 7 de junho de 2017, 2017.

[Wal17] Walber. Precision and recall. Disponível em https://en.wikipedia.org/wiki/Precision_and_recall#/media/File:Precisionrecall.svg, acedido a última vez em 10 de fevereiro de 2017, 2017.

[WSR16] Erik Wittern, Philippe Suter e Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 351–361, 2016. URL: http://dl.acm.org/citation.cfm?doid=2901739.2901743, doi:10.1145/2901739.2901743.

[ZXZ+09] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei e Hong Mei. MAPO: mining and recommending api usage patterns. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5653 LNCS, pages 318–343, 2009. doi:10.1007/978-3-642-03013-0_15.

[ZZM14] Jiaxin Zhu, Minghui Zhou e Audris Mockus. Patterns of folder use and project popularity. *ESEM conf.*, pages 1–4, 2014. URL: `http://www.scopus.com/inward/record.url?eid=2-s2.0-84907846797&partnerID=tZOtx3y1`, doi:10.1145/2652524.2652564.

Anexo A

Neste anexo é apresentado o código desenvolvido para testar os *parsers da tabela 2.1.*

A.1 Ficheiro principal de execução do código

```
const fs = require("fs");
const bs = require("benchmark");

// parsers for test
const esprima = require("esprima");
const babylon = require("babylon");
const espree = require("espree");
const acorn = require("acorn");

// Libs for test
const libs = "./libs/";
const angular = fs.readFileSync(libs+"angular.js", "utf8");
const jQueryMobile = fs.readFileSync(libs+"jquery.mobile.js", "utf8");
const react = fs.readFileSync(libs+"react.js", "utf8");

let suite = new bs.Suite;

// add tests
suite.add('esprima - angular', function() {
    esprima.parse(angular);
})
    .add('esprima - jqueryMobile', function() {
        esprima.parse(jQueryMobile);
    })
    .add('esprima - react', function() {
        esprima.parse(react);
    })
    // acorn
    .add('acorn - angular', function() {
        acorn.parse(angular);
    })
```

```
    .add('acorn - jqueryMobile', function() {
        acorn.parse(jQueryMobile);
    })
    .add('acorn - react', function() {
        acorn.parse(react);
    })
    //babylon
    .add('babylon - angular', function() {
        babylon.parse(angular);
    })
    .add('babylon - jqueryMobile', function() {
        babylon.parse(jQueryMobile);
    })
    .add('babylon - react', function() {
        babylon.parse(react);
    })
    //
    .add('espree - angular', function() {
        espree.parse(angular);
    })
    .add('espree - jqueryMobile', function() {
        espree.parse(jQueryMobile);
    })
    .add('espree - react', function() {
        espree.parse(react);
    })
    // add listeners
    .on('cycle', function(event) {
        console.log(String(event.target));
    })
    .on('error', function (event) {
        console.log(String(event.target) + ' failed');
    })
    .on('complete', function() {
        console.log('Fastest is ' + this.filter('fastest').map('name'));
        this.forEach(function (item, index) {
            debugger;
            console.log("id: "+item.name + " stats: "+item.stats.mean*1000 +"(+-"+
                (item.stats.deviation/item.stats.mean)*100 +"%)");
        });
    }).run();
```

A.2 Ficheiro de configuração

```
{
  "name": "",
```

```
  "version": "1.0.0",
  "description": "Speed test for parser libs",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node main.js"
  },
  "keywords": [
    "speed",
    "test"
  ],
  "author": "antoniocsoares",
  "license": "",
  "dependencies": {
    "acorn": "^4.0.11",
    "babylon": "^6.15.0",
    "benchmark": "^2.1.3",
    "espree": "^3.4.0",
    "esprima": "^3.1.3"
  }
}
```

Anexo B

Neste anexo são apresentado os ficheiros de teste manual para cada projeto avaliado.

B.1 Projeto 1

```
{
  "topics" : {
    "ecmascript": 5,
    "found": ["backend", "frontend", "desktop"]
  },
  "libs" : {
    "all": ["react", "react-dom", "electron", "path", "url", "net", "app"],
    "ex": ["react", "react-dom", "electron"]
  }
}
```

B.2 Projeto 2

```
{
  "topics" : {
    "ecmascript": 5,
    "found": ["backend"]
  },
  "libs" : {
    "all": ["fs", "stream", "util", "polyfills", "legacy-streams",
      "constants", "child_process", "tap"],
    "ex": ["fs", "util", "tap", "stream", "constants"]
  }
}
```

B.3 Projeto 3

```
{
  "topics" : {
    "ecmascript": 5,
    "found": ["backend", "frontend", "mobile"]
  },
  "libs" : {
    "all": ["framework7", "framework7-vue", "axios", "lodash",
      "moment", "vue", "vue-i18n", "vuex", "es6-object-assign",
      "es6-promise", "mutation-types", "mutations", "actions",
      "network", "find", "app", "getters", "en_us", "zh_cn",
      "storecache", "auto", "store"],
    "ex": ["axios", "framework7", "framework7-vue", "lodash",
      "moment", "es6-object-assign", "es6-promise", "vue",
      "vue-i18n", "vuex"]
  }
}
```

B.4 Projeto 4

```
{
        "topics" : {
    "ecmascript": 6,
    "found": ["frontend", "mobile"]
  },
  "libs" : {
                "all": ["react-native", "ramda", "bluebird",
                        "superagent-bluebird-promise", "redux", "immutable",
                        "repo", "react-redux", "configure", "redux-logger",
                        "redux-thunk", "superagent", "react", "babel-eslint",
                        "babel-preset-react-native-stage-0", "app", "constants",
                        "native", "request", "reducer", "immutable-js"],
                "ex": ["babel-eslint", "babel-preset-react-native-stage-0",
                        "bluebird", "immutable-js", "ramda", "react",
                        "react-native", "react-redux", "redux",
                        "redux-logger", "redux-thunk", "superagent",
                        "superagent-bluebird-promise"]
        }
}
```

B.5 Projeto 5

```
{
  "topics" : {
    "ecmascript": 5,
    "found": ["frontend", "canvasapp"]
  },
  "libs" : {
    "all": [],
    "ex": []
  }
}
```